Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

# ОРГАНИЗАЦИЯ ВВОДА И ВЫВОДА НА ПАРАЛЛЕЛЬНЫХ ПОТОКАХ

Отчёт о лабораторной работе № 2

по дисциплине «Современные средства программирования»

Выполнил: студент гр. 434-М1

Колпаков Н.А.

« » 2024 г.

Проверил: доцент каф. АСУ

Алфёров С. М.

« » 2024 г.

Томск 2024

# Оглавление

1. [Цели и задачи 3](#_bookmark0)
2. [Ход работы 4](#_bookmark0)

[Вывод 5](#_bookmark1)

## 1 ЦЕЛИ И ЗАДАЧИ

* 1. Цель работы:

Получить навыки распараллеливания задач и написания параллельных потоков для неразделяемых ресурсов.

* 1. Задания:

Дана прямоугольная матрица заданного типа. “Разделить” матрицу на подматрицы произвольным способом. Каждый поток должен обрабатывать отдельную подматрицу так, чтобы в целом решалась задача, соответствующая варианту. Искомое значение должно быть оформлено как общий ресурс для всех потоков, таким образом он будет не разделяемым.

Разрешить проблему неразделяемого ресурса при помощи симафоров или мютексов.

Проверить правильность решенной задачи вычислив искомое значение в основном потоке. Запустить программу при разном количестве столбцов и строк матрицы.

Построить графики зависимости времени обработки матрицы от количества элементов

матрицы (сложности задачи). Выявить сложность задачи, при которой параллельный

способ обрабатывает матрицу быстрее последовательного.

Вариант 5

Тип матрицы: целочисленная, однобайтные.

Искомое значение: сумма номеров столбцов, элементы которых равны нулю

## 2 ХОД РАБОТЫ

Листинг программы:

import threading

import numpy as np

import time

import matplotlib.pyplot as plt

first = True

column\_sum = 0

result\_lock = threading.Lock()

processed\_columns = None

def process\_submatrix(submatrix, col\_indices):

    global column\_sum

    global processed\_columns

    cols\_with\_zeros = np.any(submatrix == 0, axis=0).nonzero()[0]

    with result\_lock:

        for col in cols\_with\_zeros:

            if not processed\_columns[col]:

                column\_sum += col\_indices[col]

                processed\_columns[col] = True

def parallel\_processing(matrix, num\_threads):

    global column\_sum

    global processed\_columns

    column\_sum = 0

    processed\_columns = np.zeros(matrix.shape[1], dtype=bool)

    row\_chunks = np.array\_split(matrix, num\_threads, axis=0)

    col\_indices = np.arange(matrix.shape[1])

    threads = []

    for submatrix in row\_chunks:

        t = threading.Thread(target=process\_submatrix, args=(submatrix, col\_indices))

        threads.append(t)

        t.start()

    for t in threads:

        t.join()

def sequential\_processing(matrix):

    cols\_with\_zeros = np.any(matrix == 0, axis=0).nonzero()[0]

    return sum(cols\_with\_zeros)

def benchmark(matrix\_size, num\_threads):

    matrix = np.random.randint(0, 127, size=(matrix\_size, matrix\_size), dtype=np.byte)

    # Последовательная обработка

    start\_time = time.time()

    sequential\_result = sequential\_processing(matrix)

    sequential\_time = time.time() - start\_time

    # Параллельная обработка

    start\_time = time.time()

    parallel\_processing(matrix, num\_threads)

    parallel\_time = time.time() - start\_time

    assert column\_sum == sequential\_result, f"Результаты не совпадают! {column\_sum} != {sequential\_result}"

    parallel\_speed = (matrix\_size \* matrix\_size) / parallel\_time

    sequential\_speed = (matrix\_size \* matrix\_size) / sequential\_time

    return sequential\_speed, parallel\_speed

matrix\_sizes = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000,20000,30000]

sequential\_speeds = []

parallel\_speeds = []

for size in matrix\_sizes:

    seq\_speed, par\_speed = benchmark(size, 4)

    sequential\_speeds.append(seq\_speed)

    parallel\_speeds.append(par\_speed)

    if seq\_speed < par\_speed and first:

        plt.scatter(size, par\_speed, color='red', label=f"Размер матрицы, где параллельное быстрее ({size})")

        first = False

    print(f"Матрица {size}x{size}: Скорость последовательной обработки: {seq\_speed:.2f}, Скорость параллельной обработки: {par\_speed:.2f}")

plt.plot(matrix\_sizes, sequential\_speeds, label='Последовательная обработка')

plt.plot(matrix\_sizes, parallel\_speeds, label='Параллельная обработка (4 потока)')

plt.xlabel('Размер матрицы')

plt.ylabel('Скорость (элементы/секунды)')

plt.title('Сравнение скорости обработки матриц в зависимости от их размера')

plt.legend()

plt.grid(True)

plt.show()

Результат выполнения работы представлен на рисунке 1.1.

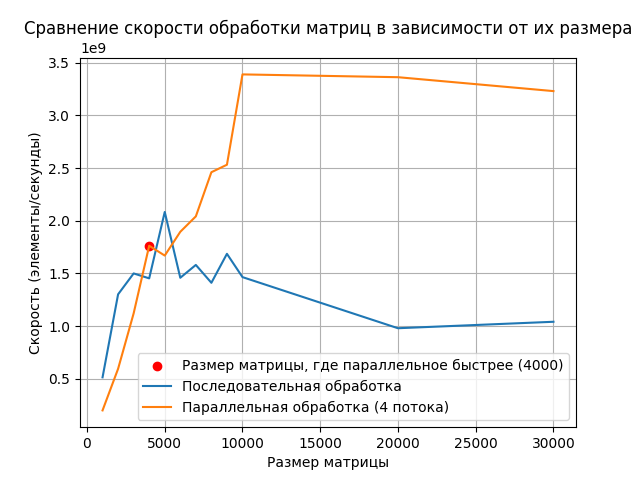


Рисунок 1.1 — Результат работы программы

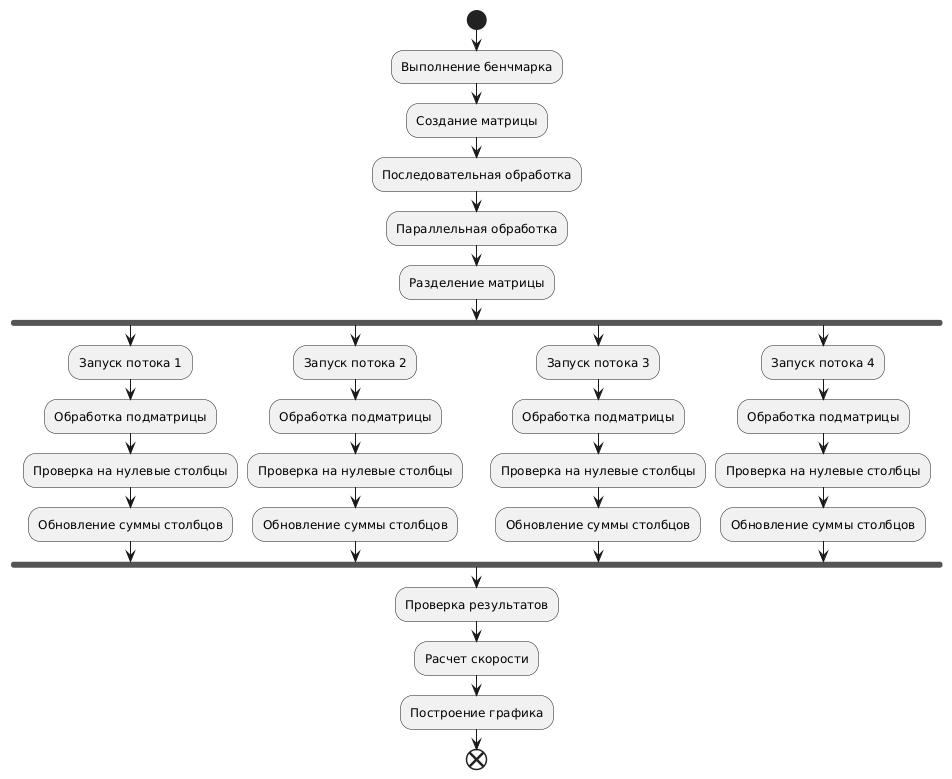


Рисунок 1.2 — UML диаграмма классов

## Вывод

В данной ходе работы исследовалась производительность параллельной обработки целочисленной однобайтной матрицы для нахождения суммы индексов столбцов, элементы которых равны нулю. Матрица была разделена на подматрицы, обрабатываемые в отдельных потоках, с использованием мьютексов для синхронизации доступа к общей переменной. Тестирование показало, что при небольших размерах матрицы последовательная обработка была быстрее, однако с увеличением размера матрицы параллельная обработка демонстрировала значительно лучшие результаты. Таким образом, лабораторная работа подтвердила, что параллельные алгоритмы эффективнее для задач с большим объемом данных, что делает их предпочтительными в вычислительных приложениях.